Pooh User's Manual

19 July 1982

Telle Whitney
Tom Hedges

Technical Report 5029

Computer Science Department
California Institute of Technology
Pasadena, California 91125

Silicon Structures Project

Table of Contents

1. Introduction 1
2. Pooh Leaf Cell Description 1
    2.1. "NotGat" Description 2
    2.2. Pooh Data Declarations 5
    2.3. Path and Contact Types 5
    2.4. Commands Available For Defining Cells 8
    2.5. Node Commands 9
        2.5.1. Points and Contacts 10
        2.5.2. Side Contacts 11
        2.5.3. Big Contacts 12
        2.5.4. Big Side Contacts 12
        2.5.5. Indirect Nodes 13
        2.5.6. Miscellaneous 14
3. Pooh Composition Cell Description 15
    3.1. Pooh Data Declarations 18
    3.2. Ports 23
    3.3. Pooh Composition Commands 26
    3.4. Moving Ports 28
4. Pooh Status Commands 30
    4.1. Intermediate Files 30
    4.2. Design Rules 30
    4.3. Response File 31
    4.4. Mossim Information 32
    4.5. Pooh Cell Information 33
5. Using Pooh On The 20 33
Index 35

List of Figures

Figure 2-1: Not Gate 2
Figure 2-2: Pooh Code 2
Figure 2-3: nMOS contacts 5
Figure 2-4: cMOS SOS contacts 6
Figure 3-1: Incrs Cell 18
Figure 3-2: Incr Pooh Code 18
Figure 3-3: Incr Leaf Cell 18
Figure 3-4: Port Definitions for Leaf Cell 18

1. Introduction

Pooh is an embedded language symbolic design system. The pooh system is a set of procedures in the programming language Mainsail whose purpose is to aid in the definition of cells. There are two types of cells: leaf cells and composition cells. A leaf cell is a cell which contains the interconnection of primitive structures and no references to other cells. In pooh, the primitive structures are wires, transistors, and contacts. These primitives provide connectivity and circuit information about the leaf cell and can be used to automatically generate geometry. A composition cell is a cell which contains instances of other cells, and their interconnections.

Currently, the pooh embedded language system allows a pooh design description to be created, and then written out to either one of two intermediate formats. The cell may either be written to a CIF file, or to an ELF file. ELF is a new higher level intermediate form, whose structure maps easily into pooh. Pooh currently supports both nMOS with buried contacts, and cMOS SOS.

Pooh was initially developed as a stand alone embedded language design system. It was then adopted as the base representation for the Silicon Compiler project developed by the Silicon Structures Project (SSP) at Caltech. There are now many status commands available whose purpose is to aid in the development of the compiler. These commands are available to any Pooh user.

This paper is divided into three section: 1) The Leaf Cell System Description, and 2) The Composition Cell System Description, 3) Pooh Status Commands Description.

2. Pooh Leaf Cell Description

This Section is divided into five parts: 1) a brief explanation of how to use pooh followed by an example, 2) the Mainsail class declarations for the primitive pooh data structures, 3) the path and contact types for both technologies, 4) the commands available for defining a leaf cell, and 5) commands available for adding nodes to paths.

Pooh allows a designer to specify a sequence of transistors, wires contacts, and symbols and to collect these together in a leaf cell. In order to use pooh, a designer must understand the concept of a path and a node. There are several different types of nodes, but a node always represents a line segment and a point in 2-dimensional space. Paths also come in various types and represent a centerline through a series of nodes with a radius. The radius is half the width of a path. Using this notion of nodes and paths, a leaf cell is easily defined - wires are paths, contacts are nodes and transistors are either paths or nodes. Symbols are a collection of paths and nodes which the designer may wish to place many times. A symbol is not a cell in that it is not a logical entity, it is a shorthand notation for reusing the same set of geometry many places.

A node has two important attributes: placement and type. In the pooh

system, placement is either direct or indirect. The direct case provides absolute placement coordinates x,y for the node. The indirect case provides other constructs on which the node depends. An indirect node is either created as the intersection of two line segments, or it is placed some specified distance from another node. A node type is one of the following: 1) a point, 2) a contact, 3) a big contact or 3) a transistor. A point is a geometric point of interest. A contact is a connection between at least two paths. A big contact is a contact with more than one connection points. A transistor is a gate caused by the intersection of two paths.

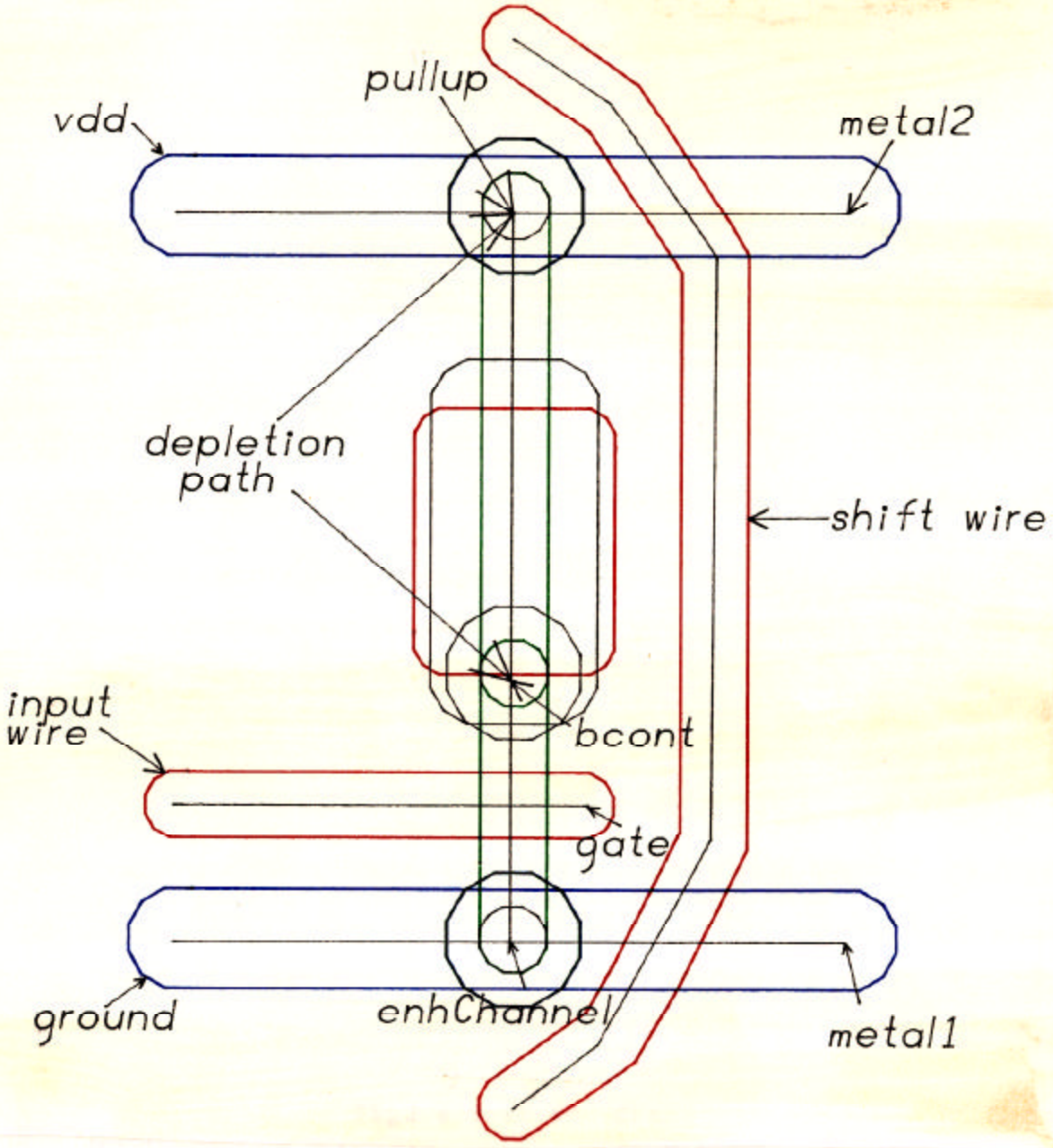
A user defines a path by first creating an empty path, and then defining and adding segments to this path by using the node definition commands. For each new segment, pooh creates and returns a node to the user, and adds a corresponding line segment definition to the path. The node, which the user may store in a variable, references both the newly created line segment plus the point at the end of the segment. The user's interface to a line segment is through the node. The line segment references the node, but contains additional information. This additional information includes the distance from the centerline to the node; when zero, the path goes through the node, when negative it goes around the node in a clockwise direction, and when positive, the path goes around the node in a counter-clockwise direction. Pooh ensures that a non-zero distance is the design rule correct distance between a node and a path.

Nodes are an important construct to a user. Since each node definition command returns a node, a user may store this node in a variable. Then, when another path connects to a previously defined node, it may reference the original node. Thus the idea of connectivity, shared nodes, is easily represented in the pooh data structure.

Figure 2-1 is a plot of the geometry generated from a pooh description. The nodes and paths are marked. Figure 2-2 is the pooh code which generated this picture. The following is an explanation of the example's statements.

2.1. "NotGat" Description

The first two newP statements, in the example, generate the cell's ground and VDD wires. The ground line segment is saved in the variable metal1, and the VDD line segment is stored in the variable metal2. The next two paths created are the channel and the gate of the pulldown, respectively. The gate is created by adding an indirect node to the channel, with the additran statement. The system ensures that the minimum overlap rules are met. The pullup statement creates a depletion mode transistor starting at the node bcont and going to the metal2 bus. Note that addinode adds an indirect node caused by the intersection of the depletion transistor and the VDD wire. The last path runs the length of the cell, missing several potentially design rule hazardous nodes. The last path makes use of the addinxy, which creates an indirect node offset the specified amount from a previously defined node, in this case the indirect node pullup.


pullup
vdd
metal2
depletion
path
shift wire
input
wire
bcont
gate
ground
enhChannel
metal1

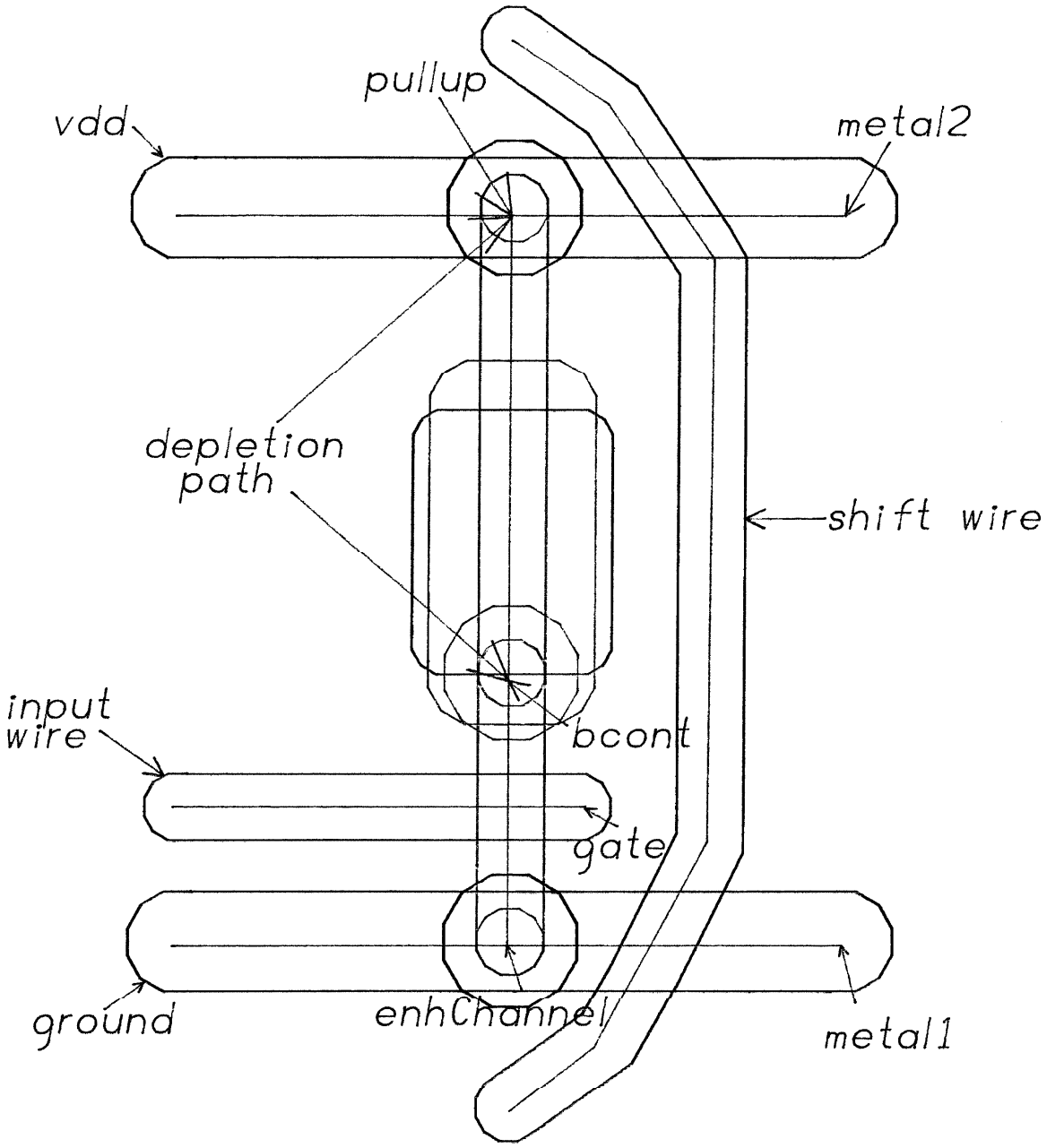


Figure 2-1: Not Gate

```
Begin "notgat"

#get the design rule definitions
sourcefile "poohnmos.h";

initial procedure;
begin
    #mainsail constants just for ease
    define metally=0., metal2y=22., leftx=0.,rightx=20.;

    pointer(node)enhChannel,toppull,gate,metal1,metal2,bcont;
    pointer(cellDesc)notg;

    #set the technology the design is in
    setnmos;

   #set the output file
   openoutputfile("inverter",usecif!useelf);

    #define an inverter
    definecell("not");

        #create ground metal bus
        newp(metal); axy(leftx,metally); metal1:=ax(rightx);

        #create VDD bus
        newp(metal); axy(leftx,metal2y); metal2:=ax(rightx);

        #diffusion part of pulldown
        newp(diff);bcont:=axy((rightx-leftx)/2.,metally+8.,pd);
        enhChannel:=addinode(metal1,dm);

        #pullup
        pullup(bcont,8.); toppull:=addinode(metal2,dm);

        #poly part of pulldown
        newp(poly); axy(leftx,4.); gate:=additran(enhChannel,enh);

        #shift wire that runs around the cell
        newp(poly); addinxy(toppull,0.,5.);
            addpt(toppull,-1.);
            addinxy(toppull,5.,-2.,noc,-1.);
            addpt(gate,-1.); addpt(enhChannel,-1.);
            addinxy(enhChannel,0.,-5.);

    #finish up the cell, and write the description to a file
    notg:=enddefine;

end;

end  "notgat"
```

Figure 2-2: Pooh Code

2.2. Pooh Data Declarations

The following are the class declarations of the useful pooh data types. Attributes of these data structures may be accessed by a designer:

```
CLASS(thing)path (INTEGER ptype;                    #path type
                  POINTER(segment) ARRAY (0 to *)ListSegs;
                                                    #list of Line Segments
                  REAL radius;                      #radius of the path
                 );

CLASS(thing)node (REAL x,y;                        #absolute coordinates
                  INTEGER contype,                 #if contact, contact type
                  STRING name;                     #name of the node
                 );

CLASS symbol     (
                  INTEGER cifcellIndex,            #index to cif cell
                          elfcellIndex;            #index to elf cell
                  STRING name;                     #name of symbol
                 );

# transform - a graphical transformation
CLASS transform(REAL t11,t12,t21,t22,tx,ty; #the coordinates of a
                                            #graphical
                                            #transformation matrix
            POINTER(transform) last;        #pointer to the last
                                            #transfrom on the
                                            #stack
            );
```

2.3. Path and Contact Types

Pooh supports both nMOS and cMOS. The following describes the predefined types available. Figure 2-3 are the nMOS contact, and figure 2-4 are the cMOS contacts. These are the node and path types available for both technologies:

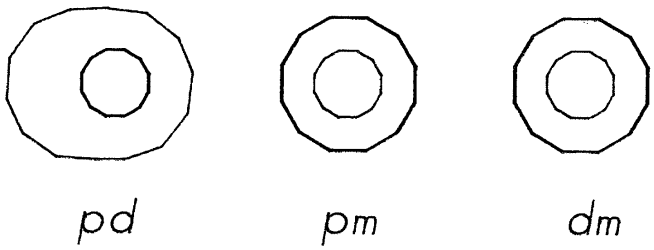


Figure 2-3: nMOS contacts

NMOS contact types

pd - poly to diffusion
dp - poly to diffusion
pm - poly to metal
mp - poly to metal
dm - diffusion to metal
md - diffusion to metal
noc- no contact

path types

depd - depletion mode transistor, diff centerline
depp - depletion mode transistor, poly centerline
enhp - enhancement mode transistor, poly centerline
enhd - enhancement mode transistor, diff centerline
diff - diffusion
poly - polysilicon
metal - metal
zzz - no path type

Indirect Transistor Types

dep - depletion mode transistor
enh - enhancement mode transistor

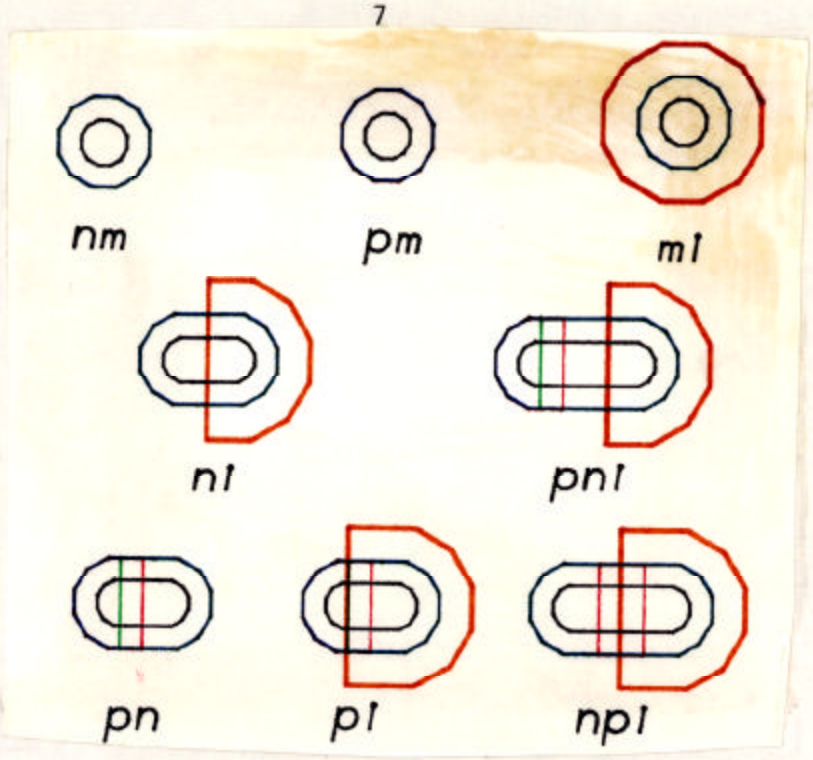


Figure 2-4: cMOS SOS contacts

CMOS SOS contact types

nm - n-island to metal
mn - n-island to metal
pm - poly to metal
mp - poly to metal
mi - metal to p-island
im - p-island to metal
noc - no contact

big contact types - more than one connection - the order of the layer in the type definition name is indicative of the order of the connection points

npi - n-island connects to connection point 1
poly connects to connection point 2
p-island connects to connection point 3
pni - poly to n-island to p-island
pn - poly to n-island
ni - n-island to p-island
pi - poly to p-island

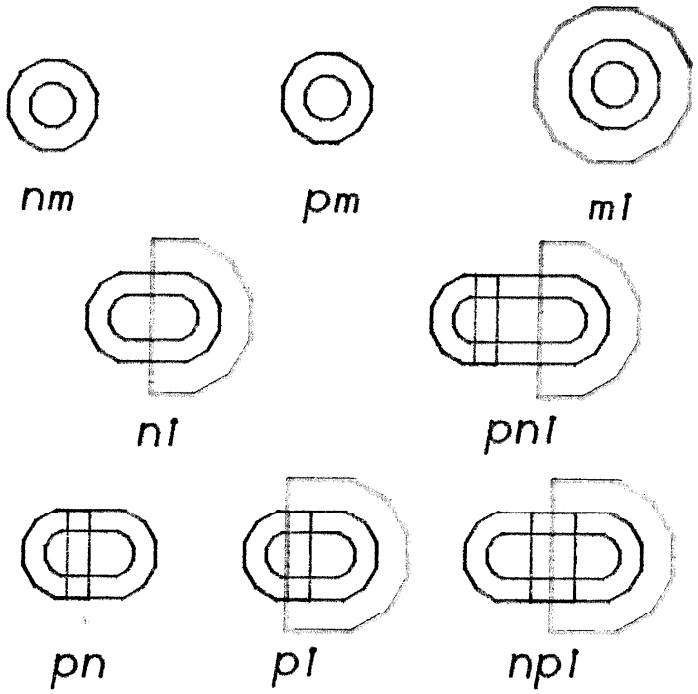


Figure 2-4: cMOS SOS contacts

CMOS SOS contact types

nm - n-island to metal
mn - n-island to metal
pm - poly to metal
mp - poly to metal
mi - metal to p-island
im - p-island to metal
noc - no contact

big contact types - more than one connection - the order of the layer in the type definition name is indicative of the order of the connection points

npi - n-island connects to connection point 1
poly connects to connection point 2
p-island connects to connection point 3
pni - poly to n-island to p-island
pn - poly to n-island
ni - n-island to p-island
pi - poly to p-island

```
path types

ptypi   - p type of transistor, p-island centerline
ptypp   - p type of transistor, poly centerline
ntypi   - n type of transistor, n-island centerline
ntypp   - n type of transistor, poly centerline
islandn - n-island
islandp - p-island
poly    - polysilicon
metal   - metal
zzz     - no path type

Indirect Transistor Types

ptyp   - p type of transistor
ntyp   - n type of transistor
```

2.4. Commands Available For Defining Cells

This section describes the procedures available for creating a path, creating a cell, and creating a symbol.

A path has an optional radius which is half the width of the path. The radius, if omitted, defaults to the minimum radius of the path type. For a wire, this is obvious. For a transistor, this number denotes the radius of the centerline layer. The length of a transistor is another optional parameter which, if present, indicates the length of the transistor measured along the centerline. The ratio of the transistor is length/(2*radius). If a length is specified by a designer, only the centerline layer will continue after the length has been achieved.

NewP — Create and return a new wire of type ptype.

```
pointer(path) procedure newP(integer ptype;
optional real radius);
```

NewT — Create and return a new transistor of type ptype.

```
pointer(path) procedure newT(integer ptype;
optional real length,radius);
```

DefineCell — Start the definition of a cell or a symbol with the given name. Cell definitions may be nested, with no loss of information.

```
procedure definecell (string name);
```

EndDefine — End the definition of a cell. Write the internal cell definition to an intermediate file. Return a pointer which describes external connections, and provides an index into the intermediate file. The nodrc flag indicates that no design rule checking is to be performed. Currently, design rule checking is not implemented, but soon will be. The erflag indicates whether a pooh error has been found in the cell.

```
pointer(cellDesc) procedure enddefine(
produces optional boolean erflag;
optional boolean nodrc);
```

EndSymbol — End the definition of a symbol. Return a pointer which describes the topology of the symbol. The nodrc and erflag flags have the same definition as in the 'endDefine' command.

```
pointer(symbol) procedure endsymbol(
produces optional boolean erflag;
optional boolean nodrc);
```

PlaceSymbol — Place a symbol according to the specified transformation. The procedure returns an array trpts of all the transformed symbol nodes. Instancename is the name of this instance of the symbol.

```
procedure placesymbol(pointer(symbol)s;
pointer(transform)tr;
optional string instancename; optional
produces pointer(node) array(1 to *) trpts);
```

2.5. Node Commands

A path is defined by specifying an ordered series of nodes which determine the centerline of the path. The command used to create the node indicates the node type; the type is not an explicit parameter. There are several different sets of commands for creating nodes with different attributes: point and contact commands, bigcontact commands, sidecontact and big sidecontact commands, and indirect node commands. All of the following commands, except where noted in the miscellaneous commands, return a node.

Points are geometric points, of interest strictly for placement of a path. Contacts are connections between at least two layers with only one connection point. Bigcontacts are contacts with more than one connection points, currently available only in cMOS. Side contacts are contacts that connect to the side of a path. Big Sidecontacts are Side contacts where the contact is a big contact. Indirect nodes are nodes created as a result of either a displacement from another node, or the intersection of two paths. An indirect node may be any one of the

previously defined node types, or it may be a transistor, caused by the intersection of the channel and the gate.

The final set of procedures are miscellaneous commands. One command allows the designer to associate an existing node with an existing path for connectivity purposes. This command should always be used if the node is connected to a path, but was not used in the definition of the path. Another command allows the designer to associate a name with a node. Finally there are procedures which are technology dependent. These are procedures which generate structures which may contain design rules violations internally, but are okay in the given technology. In particular there is a pullup procedure for nMOS.

The distance, where applicable, is the distance the path centerline travels relative to the node. Positive distance is counter-clockwise; negative distance is clockwise. Pooh ensures that a non-zero distance meets the design rules. The default distance, if omitted, is zero.

2.5.1. Points and Contacts

Points and contacts are created with the same set of procedures. The node is a contact if a contact type is specified. If the contact type is 'noc', or the type is omitted, the node is a point.

Both the Relative and Absolute Distance procedures have the following format:

```
pointer(node)procedure <pname>(<non optional parameters>,
      optional integer contype; optional real distance);
```

Contype is the type of the contact, distance is the distance of the path from the node.

Relative Distance (Non Optional Parameters Shown)

`dx(real dx);`	Move relative to the last node in the x direction.
`dy(real dy);`	Move relative to the last node in the y direction.
`dxy(real dx,dy);`	Move relative to the last node in both x and y.

Absolute Distance (Non Optional Parameters Shown)

`ax(real x);`	Move to this x coordinate, and use the last node's y.

```
ay(real y);        Move to this y coordinate, and use the
                   last node's x.

axy(real x,y)      Move to this x and y.
```

PreDefined Node

```
addpt(optional pointer(node)p; optional real distance);
                   Move to a node already defined. If the
                   node is not present then the last
                   defined node is used.
```

2.5.2. Side Contacts

One connection Side Contacts are created with the next set of procedures. There are no optional parameters. Contype is the contact type. Dir is the direction the path moves around the node, -1 is clockwise, +1 is counterclockwise.

Relative Distance

```
sdx(real dx; integer contype,dir);
                   Move relative to the last node in the x
                   direction.

sdy(real dy; integer contype,dir);
                   Move relative to the last node in the y
                   direction.

sdxy(real dx,dy; integer contype,dir);
                   Move relative to the last node in both x
                   and y.
```

Absolute Distance

```
sax(real x; integer contype,dir);
                   Move to this x coordinate, and use the
                   last node's y.

say(real y; integer contype,dir);
                   Move to this y coordinate, and use the
                   last node's x.

saxy(real x,y; integer contype,dir);
                   Move to this x and y.
```

Predefined Contact

```
saddpt(pointer(node)p; integer dir);
                    Move to this predefined node.
```

2.5.3. Big Contacts

The following commands are used to create big contacts, contacts with greater than one contact point, currently available only in cMOS. Pindex denotes the connection point desired for this path. The values for x and y define the position of the first contact point. The degree parameter specifies in degrees the orientation of the contact, where 0 degrees places the last connection point collinear to the first point directly along the positive x-axis. Contype and dis are the contact type and distance.

Relative Distance

```
bdx(real  dx;  integer  degrees,pindex,contype;optional
                    real dis);

bdy(real   dy;  integer  degrees,pindex,contype;optional
                    real dis);

bdxy(real dx,dy; integer degrees,pindex,contype;optional
                    real dis);
```

Absolute Distance

```
bax(real x; integer degrees,pindex,contype;optional real
                    dis);

bay(real y; integer degrees,pindex,contype;optional real
                    dis);

baxy(real x,y;  integer  degrees,pindex,contype;optional
                    real dis);
```

Predefined Big Contact

```
baddpt(pointer(node)p;integer   pindex;   optional   real
                    dis);
```

2.5.4. Big Side Contacts

The following commands are used to create big contacts which the current path contacts along the side of the path. The degree parameters specifies in degrees the orientation of the contact, where 0 degrees places the last connection point collinear to the first point directly along the positive x-axis. Pindex is which point the path references (it must be less than the number of connection points). Contype is the

contact type. Dir is direction the path moves around the contact, +1 is counter clockwise, -1 is clockwise.

Relative Distance

```
sbdx(real dx; integer degrees,pindex,contype,dir);

sbdy(real dy; integer degrees,pindex,contype,dir);

sbdxy(real dx,dy; integer degrees,pindex,contype,dir);
```

Absolute Distance

```
sbax(real x; integer degrees,pindex,contype,dir);

sbay(real y; integer degrees,pindex,contype,dir);

sbaxy(real x,y; integer degrees,pindex,contype,dir);
```

Pre-Defined Big Contact

```
sbaddpt(pointer(node)p;integer pindex,dir);
```

2.5.5. Indirect Nodes

This next set of commands create indirect nodes. The nodes used to create the desired line segment(s) are the required parameters to the procedures which create a new node at the intersection of the two segments.

Inode — Create a point or a contact at the point of intersection of p1 and p2. If contype is 'noc', then a point is created, otherwise the node is a contact.

```
inode(pointer(node)p1,p2;
    optional integer contype);
```

Addinode — Create a point or a contact at the intersection of the current active path and p. If contype is 'noc', then a point is created, otherwise the node is a contact.

```
addinode(pointer(node)p; optional
integer contype;optional real dis);
```

Tran — Create a transistor at the intersection of channel and

gate of type trantype.

```
tran(pointer(node)channel,gate;
integer trantype);
```

AddiTran — Create a transistor at the intersection of the current active path, and channel. The system will ensure that minimum overlap rules for the transistor are met.

```
additran(pointer(node)channel; integer ptype);
```

AddInxy — Create a new indirect point or contact offset from an old node the specified x and y. P is the node this new node is offset from. Contype is the contact type if specified, and dis is the distance the path is to move relative to the new node.

```
addinxy(pointer(node)p;real x,y;
optional integer contype; optional real dis);
```

Binode — Create a big contact whose first connection point is placed at the intersection of p1 and p2.

```
binode(pointer(node)p1,p2;
    integer degrees,pindex,contype);
```

Baddinode — Create a Big Contact whose first connection point is at the intersection of the current active path and p.

```
baddinode(pointer(node)p;integer
degrees,pindex,contype; optional real dis);
```

2.5.6. Miscellaneous

Addnode — Add an existing Node to the Description of an existing line segment p, for connectivity purposes. Note that an arbitrary number of nodes may be passed in, and that a node is not returned.

```
procedure addnode(pointer(node)p;
repeatable pointer(node)ptn);
```

Name — Associate the string nname with the node p. If the node is not specified, then name the most recently defined

node. The node is not returned.

```
procedure name(string nname;
optional pointer(node)p);
```

Pullup — Create a pullup starting at the node nl of length leng. Nl should be a buried contact, or pooh gives an error message. Pooh returns the buried contact. Note this procedure is only available in nMOS.

```
pointer(node)procedure pullup(pointer(node)nl;
real leng; optional real radius);
```

PullupXY — Create a pullup starting at x,y of length leng. Pooh returns the buried contact. Note this procedure is only available in nMOS.

```
pointer(node)procedure pullupxy(real x,y;
real leng; optional real radius);
```

3. Pooh Composition Cell Description

The Pooh composition system allows a designer to connect instances of other cells. Each time a cell is designed, a user designates some nodes as ports. Ports are points that are accessible from outside the cell definition. A composition cell is a series of instances of other cells whose ports are glued together. Composition cells in Pooh do not necessarily have to be purely composition cells, that is they may also contain geometry (paths and transistors). Pooh does not know how to stretch cells, so to connect two instances, the designated ports of the two instances must line up exactly and must be "compatible".

A composition cell definition consists of a series of compose commands. Each compose command composes an instance of a previously defined cell definition with the current composition context, and then creates a new composition context. Initially the first composition context will be a single cell instance. Then instances will be glued down, until finally the cell definition is complete.

Each port, when it is created, is designated as belonging to a side of the cell. Two instances are glued together by gluing two sides together. There are four predefined sides: 1)North, 2) South, 3) West and 4) East. Their intended use is to define ports along the top, bottom, left, and right of a cell respectively, though this is by no means enforced by pooh. The user may define additional sides as is necessary.

The procedures of the composition system are always called in a specific order, with some calls required and others optional. The first call for a new composition cell is 'initialCell' which places the first

instance and specifies its transformation, including translation. Then there are optional calls to modifier procedures which are used to manipulate the ports from the instance. Finally, 'executeComposition' is called to finish this cell's addition to the current composition context. Note that for the initial cell there was nothing previously in the current composition context. To continue the composition process with other cells the procedure 'compose' is used, rather than 'initialCell', but the modifier procedures and 'executeComposition' are used in the same way. Compose takes additional arguments which specify the side of the current context and the side of the new cell to mate.

The modifier procedures, flushPort, omitPort, keepPort, connectPort, mapSide and assignSignal, use a common scheme for 'addressing ports' or determining which of the current set of ports to which they are referring. The most significant parameter is a boolean which is true if a port from the cell being added is being addressed and is false if a port from the current context is being addressed. Note that during an initialCell command, only the true option is meaningful since the composition context is empty at that point. The next most significant parameter is the side - only those ports that are on the designated side are considered further. Then comes the internal name parameter. The pooh system finds the port (or ports) whose internal name matches exactly the internal name parameter (case is ignored). If an '*' is present in the internal name, it will match zero or more arbitrary characters. Only one '*' per internal name is allowed. If there is exactly one port that matches then that port is selected and position is not considered. If there are multiple ports that satisfy all the conditions, then the position parameter is used to break the tie.

Position may be used in three rather different ways: First, if position is positive, then it is assumed to be an arbitrary user-defined index that is to be matched exactly and all the ports under consideration are tested and if exactly one matches then it is selected, otherwise an error is reported. Second, if the default sides are being used for the ports under consideration, then relative geometric positioning may be specified. There are four predefined relative positions: northMost, eastMost, southMost, and westMost. For each side the bordering side's directions are available (e.g. on the east, northMost and southMost). A relative position expression may be specified by either the relative direction alone (meaning the end port in the row) or by the relative direction minus an integer. For example if the east side has five ports with the same internal name, they could be referred to as northmost, northmost-1, northmost-2, northmost-3, and northmost-4. Alternatively that could be southmost-4, southmost-3, southmost-2, southmost-1, and southmost; thus in this case northmost-2 is the same as southmost-2. The integer need not be a constant, of course. The meaning of the ordering is based on the coordinate locations of the ports in their original frame of reference (that is before the current transform is applied). Note also that only those ports that are on the same side and have the same internal name are considered here, not all the ports on the given side. Third, if position is equal to the predefine constant 'alwaysmatch' then pooh will process all ports on the specified side which match the internal name parameter.

FlushPort is used to remove a port from the resulting composition context (and thus not include it at the next higher level of

composition). The normal action of the system is to keep all ports except those that matched other ports during the composition.

KeepPort forces a port to be kept in the composition context, even if it matched up with another port during composition. KeepPort, like some other modifier procedures, has optional arguments that allow the port to be modified (this instance of it to be modified). The side, internal name, and position fields may be altered by specifying these parameters. KeepPort may be used on ports that would have been kept by default simply to perform the modification functions.

OmitPort causes a port to be treated as if it were not on the side being composed, even though it is. This prevents any error messages because the port did not match up and also causes the port to default to being kept. If it is desired that the port go away completely as well as not being considered for matching, then flushPort must be called for the port.

ConnectPort is the logical opposite of omitPort, namely it forces the given port to be considered as if it were on the current side. This normally causes the port to default to being flushed, since it must match up with something. KeepPort may be called if the port is to be kept.

MapSide specifies the mapping of sides between the cell being added and the sides of the context. If the default sides (north, east, south, west) are used, then the transformations will automatically generate the appropriate side mapping as a default. MapSide may be used to override this, or more likely it is required for non-standard sides since there is no default mapping of user defined sides.

AssignSignal is used to attach a signal name to a port. Signals are compared by their pointer values, not by their names. Signals should be generated with the command NewSignal to ensure that two signals with the same name are equivalent. A hash table of signal names is kept by Pooh to ensure that there is exactly one signal name record (and thus pointer value to it) for each signal name string.

When the current stage of the composition process is complete, the procedure executeComposition is called to perform checking, compute the translation factor, copy ports into the new composition context and perform any desired modification of the ports, and put out an instance reference to the cell just added. Note that if a port was modified by keepPort, flushport or omitPort the modification does not take effect until after the checking has been performed.

Checking is performed by executeComposition to ensure that the ports being connected are correctly matched. The first type of checking is based on geometric position. The smallest coordination value of each set of ports being matched is used to determine the translation factor to be applied to all the ports of the additional call. After the translation is applied, all pairs of ports are checked for exact superposition. Next layer or color is checked. Each pair of ports must either have the same layer or contain a contact which can legally connect to the other port. Last the signals are matched. For each pair of ports there are two cases: signal assigned or no signal assigned. If one port has a signal, then the mating port should have the same signal. If neither port has a

signal then the internal names and connect names are checked. If both ports have null connect names, then the internal names must match exactly (case is ignored). If either port has a connect name, then it must match the internal name of the other port and vice versa; that is both ports must have connect names and internal names that cross match. The position field is always ignored.

There is special procedure called expungePort that has a calling sequence like a modifier procedure, but which may be used either as a modifier procedure or on its own between compositions (that is after executeComposition and before another compose) to remove ports. Unlike the other modifier procedures its actions take effect immediately. Thus when it is used during the composition the ports it removes not only go away, but do not take part in the matching or alignment processes at executeComposition time (as contrasted with flushPort). It may also be used between (or after) compositions referring to the current composition context only (since the additional cell is not defined at that time) to remove ports that are not wanted and which cannot be easily gotten rid of using the standard modifier procedures.

Figure 3-1 is the plot of an example of a composition cell consisting of eight instances of a single leaf cell. Figure 3-2 is the pooh code which generated this composition cell. Figure 3-3 is a plot of the original leaf cell with the ports marked. Figure 3-4 is the port declarations from the original leaf cell.

3.1. Pooh Data Declarations

The following are the class declarations of the useful pooh composition constructs:

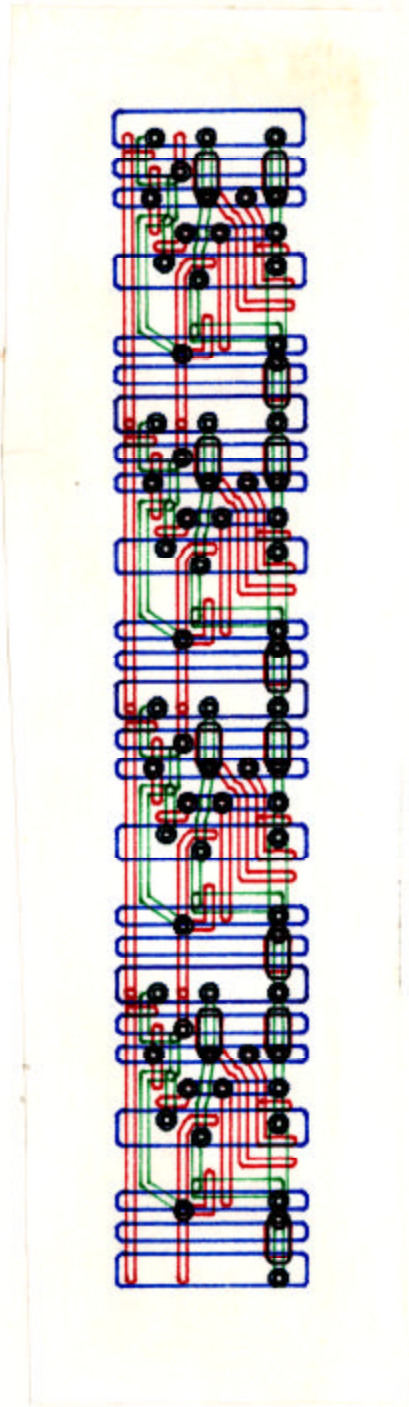


Figure 3-1: Incrs Cell

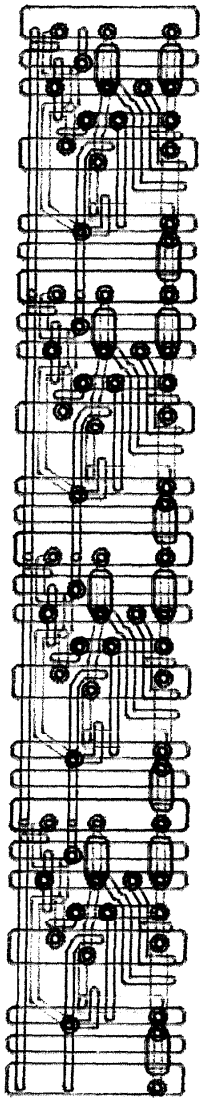


Figure 3-1: Incrs Cell

```
Begin "incrs"

sourcefile "poohnmos.h";

initial procedure;
begin
    pointer(cellDesc) incr, incrregen, incs;
    integer i;

    # start up pooh
    setnmos;  #set up the design rules
    openoutputfile("incr"); #set file name for all cells

    # compute the derived spacings
    vddtopcon := vddupper + pwrrad - 2.0;
    vdduppercon := vddupper - pwrrad + 2.0;
    vddlowercon := vddlower + pwrrad - 2.0;
    vddbotcon := vddlower  -  pwrrad + 2.0;
    gndupper := pwrrad - 2.0;
    gndlower := -pwrrad + 2.0;

    # build two types of leaf cells
    incr := incrcell("incr", false);
    incrregen := incrcell("incrregen", true);

    # composition of incrementers
    definecell("incrs");  #start composition definition
    # put the first cell down - start the composition context
    initialCell(incr, unityTransform);
    executeComposition;     #execute the defined composition
    # make an 8 bit incrementer
    for i := 1 upto 3 do
    begin
        # add another on our north side (building in y direction)
        if (i mod 4) = 0 then
            compose(incrregen, north, south, unityTransform)
         else  compose(incr, north, south, unityTransform);
        # remove extra VDD ports in the current context
        flushPort(false, "VDD", east, northmost);
        flushPort(false, "VDD", west, northmost);
        # put the cell down
        executeComposition;
    end;
    incs := enddefine;   #end the cell definition

    end;
end "incr";
```

Figure 3-2: Incr Pooh Code

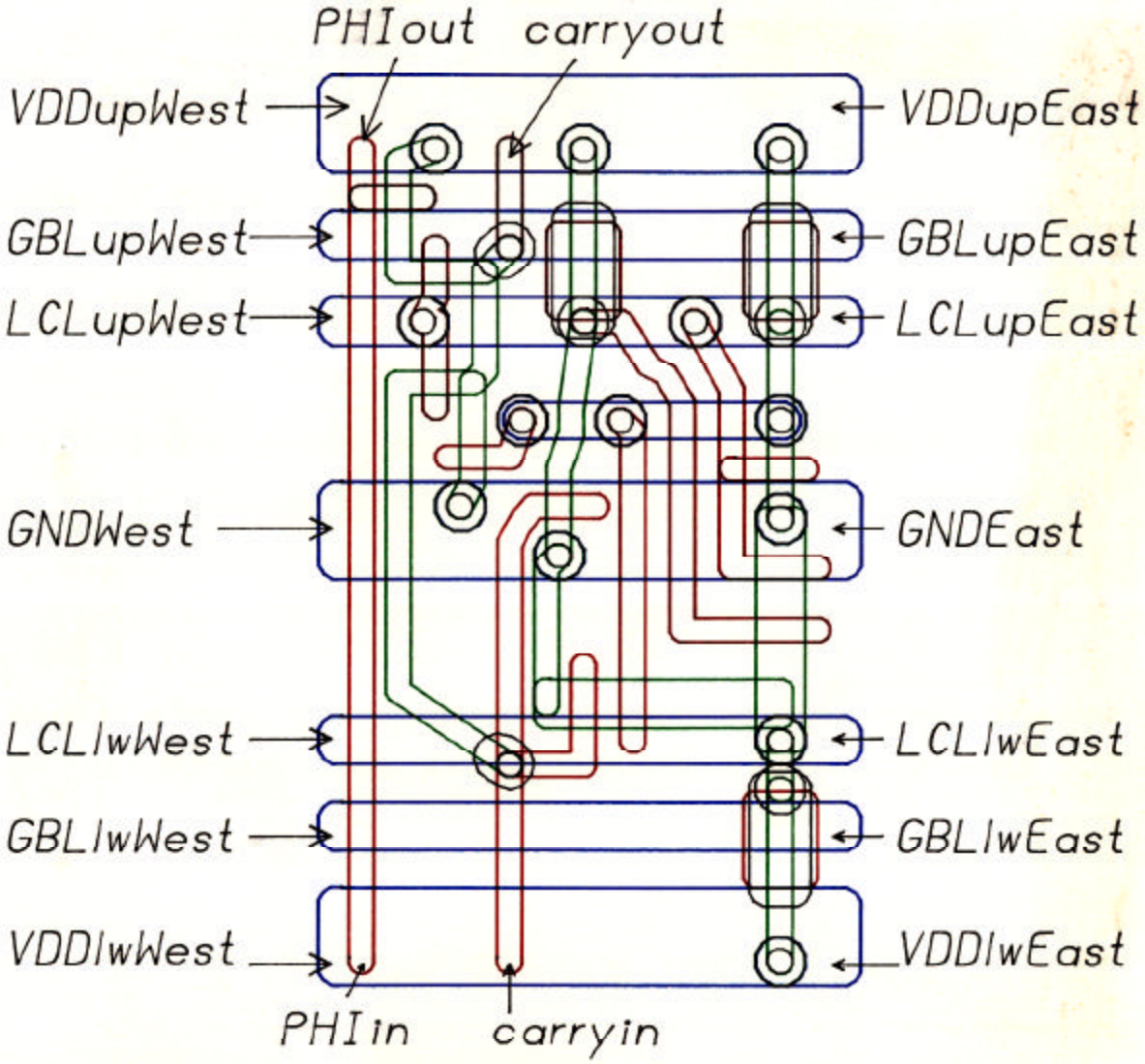


Figure 3-3: Incr Leaf Cell

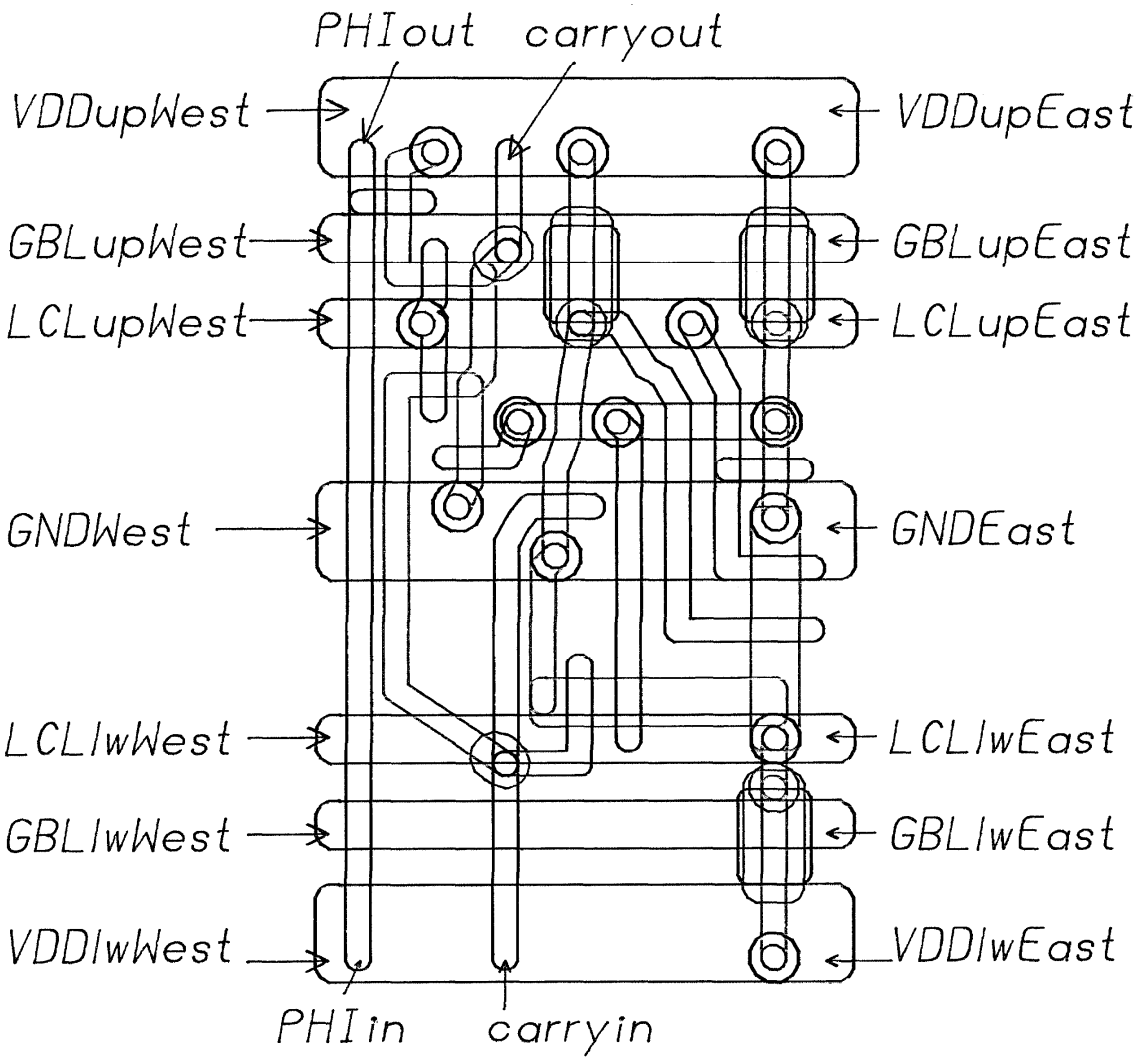


Figure 3-3: Incr Leaf Cell

```
pointer(cellDesc) procedure incrcell(string cellname; boolean regen);
begin
    # misc nodes
    pointer(node) pt, pt2, pt3, gate, diffusion, cinode;
    # contact nodes
    pointer(node) coVDD, ciVDD, inVDD,
        inL, inR, cibarUP, inbarUP, outLVDD,
        inbarL, inbarM, inbarR, GNDco, GNDorL,
        GNDorR, cibarL, cibarR,
        ciUP, outcon, outUP, carryin, carryout,
        phiIN, phiOUT, cocon, cirLVDD;
    # bus and power nodes
    pointer(node) VDDupEast, VDDupWest, GBLupEast, GBLupWest,
        LCLupEast, LCLupWest, GNDEast, GNDWest,LCL1wEast, LCL1wWest,
        GBL1wEast, GBL1wWest, VDD1wEast, VDD1wWest;
    # width of cell
    define  width=40.;


    # start the cell definition
    definecell(cellname);

    #code to generate the geometry would be here

    # add ports
    # Buses and Power
    makePort(VDDupEast, portypVDD, east, "VDD", "", VDDsignal);
    makePort(VDDupWest, portypVDD, west, "VDD", "", VDDsignal);
    makePort(VDDupEast, portypVDD, north, "VDD", "", VDDsignal);
    makePort(VDDupWest, portypVDD, north, "VDD", "", VDDsignal);
    makePort(GBLupEast, portypPreBus, east, "Upper Global Bus");
    makePort(GBLupWest, portypPreBus, west, "Upper Global Bus");
    makePort(LCLupEast, portypPreBus, east, "Upper Local Bus");
    makePort(LCLupWest, portypPreBus, west, "Upper Local Bus");
    makePort(GNDEast, portypGND, east, "GND", "", GNDsignal);
    makePort(GNDWest, portypGND, west, "GND", "", GNDsignal);
    makePort(LCL1wEast, portypPreBus, east, "Lower Local Bus");
    makePort(LCL1wWest, portypPreBus, west, "Lower Local Bus");
    makePort(GBL1wEast, portypPreBus, east, "Lower Global Bus");
    makePort(GBL1wWest, portypPreBus, west, "Lower Global Bus");
    makePort(VDD1wEast, portypVDD, east, "VDD", "", VDDsignal);
    makePort(VDD1wWest, portypVDD, west, "VDD", "", VDDsignal);
    makePort(VDD1wEast, portypVDD, south, "VDD", "", VDDsignal);
    makePort(VDD1wWest, portypVDD, south, "VDD", "", VDDsignal);
    # carry and Phi 1
    makePort(carryin, portypInput, south, "Carry In", "Carry Out");
    makePort(carryout, portypOutput, north, "Carry Out", "Carry In");
    makePort(PHIin, portypPHI1, south, "PHI1", "", PHI1signal);
    makePort(PHIout, portypPHI1, north, "PHI1", "", PHI1signal);

    # all done
    return(enddefine);
end;
```

Figure 3-4: Port Definitions for Leaf Cell

```
CLASS  signal(
                    RTRING key;          # ASCII Name of Signal
                    BITS sflags;         # signal flags
                    INTEGER stype;       # signal type
                    );

# Pooh cell descriptor
CLASS(thing) cellDesc(
                    INTEGER cifcellIndex,#index to cif cell description
                            elfcellIndex;#index to elf cell description
                    POINTER(port) ARRAY (1 to *) pports;
                                          #list of cell's ports
                    STRING name;          #name of cell
                    );

# Pooh cell instance descriptor
CLASS(thing) instance(
                    POINTER(cellDesc) cell;      # pointer to cell desc
                    POINTER(transform) trans;    # transform for instance
                    POINTER(xport) ARRAY (1 to *) pports;
                                                 # transformed ports
                    INTEGER ARRAY (north TO west) newDirections;
                                                 #side mapping
                    STRING instancename;         #name of instance
                    );

#a port
CLASS(thing) port(
                    POINTER(ptnode) pt;  # pointer to Pooh point
                    POINTER(signal) sig; # signal pointer
                    REAL x, y;           # coordinate location
                    INTEGER ptype;       # classification of port
                    INTEGER position;    # index to identical names
                    INTEGER side;        # side of cell port is on
                    REAL field1, field2; # type-dependent data
                    STRING internalname; # internal name of port
                    STRING connectname;  # name for matched connection
                    );
```

3.2. Ports

Ports are the external connections of a cell. Once a cell is defined, ports are the only information about a cell left in the pooh system. All the cell's internal description is stored in a file.

Ports are a logical entity rather than a physical one. There are several different ways a port may be named. Any given port may have one or more of the following names:

Signal — A signal is used for global signals. They describe ports that should be connected together in the same electrical net. There are four predefined signal pointers available:

VDDsignal	VDD
GNDsignal	Ground
PHI1signal	Clock Phase 1
PHI2signal	Clock Phase 2

A user may define a new signal by a call to the procedure newsignal. Sname is the signal name. The *parameters sflags and stype are* user defined. Signals are compared by pointer value internally, not by the fields of the signal record. NewSignal ensures there is exactly one signal record for each unique signal name by looking up each name in a hash table maintained internally by Pooh.

```
pointer(signal) procedure newsignal(STRING name;
BITS sflags; INTEGER stype);
```

A user may access a previously defined signal by a call to the procedure getsignal -- this procedure also looks up the signal in the internal hash table of signals, but does not modify the entry since it is assumed to already exist.

```
pointer(signal) procedure getsignal(STRING name);
```

Internal Name — an internal name is a string which designates the cell's idea of the port's name.

Connect Name — The connect name, if specified, is the internal name of ports which this port can legally connect to. For example: one port's name is carry in. Its connect name could be carry out. Therefore carry out ports may connect to carry in ports.

Besides the port's names, there are several other attributes the user may define and access:

pt	a pointer to pooh's internal description of a nodes
x,y	the physical location of the port
ptype	Port type. Port types may be one of the following:

portypVDD	VDD
portypGND	GND
portypPHI1	PHI 1

portypPHI2	PHI 2
portypInput	Input Port
portypCntlInp	Input Port that passes through (may be connected to other Input Ports)
portypOutput	simple Output Port
portypTriIO	Tristate Input/Output Port
portypTriOut	Tristate Output Port
portypBus	Bus
portypPreBus	Precharged Bus
portypUser	Any port type greater than or equal to portypUser are user defined.

Side — Which side of the cell the port is on. Predefined sides are:

north	Top side of the cell. (Positive Y)
east	Right side of the cell. (Positive X)
south	Bottom side of the cell. (Negative Y)
west	Left side of the cell. (Negative X)

Position — If there is more than one port on a side with the same name, then this field indicates which port this is. Positions may be assigned by makePort as arbitrary positive integers that are uniquely assigned to ports that share the same internal name and side. The position argument to modifier procedures may specify this position value, or may also specify relative geometric positions if a default side is used. This is described in detail above. The relative positions are:

northmost	Top Port on east/west sides. (Largest Y)
eastmost	Rightmost Port on north/south sides. (Largest X)
southmost	Bottom Port on east/west sides. (Smallest Y)
westmost	Leftmost Port on north/south sides. (Smallest X)

Field1,Field2 Parameters the user wishes to associate with a port. These would probably be electrical Parameters.

A new port within a cell definition is created by the following procedure:

```
pointer(port) procedure makePort(pointer(node) ppoint;
integer type, side; optional string internalName, connectName;
optional pointer(signal) signalName;
optional real field1, field2;
optional integer position);
```

3.3. Pooh Composition Commands

The pooh composition commands are:

InitialCell Set pcell as the initial composition context placing it according to ptrans, with the name instancename. NOTE: In the special case when cells are placed and then 'wired up' by connecting paths to all ports rather than composing, initialCell may be called more than once (still followed by modifier procedures and executeComposition). This mode of operation requires a complete transformation, including translation, being given to each placement so done. No checking of ports is performed in this case (it is never done for initialCell in any case), and no automatic translation factor is computed.

```
procedure initialCell(pointer(cellDesc) pcell;
pointer(transform) ptrans;
optional string instancename);
```

Compose Compose acell's aside to the current composition context's cside, with the transformation ptrans. Ptrans should only include mirroring and rotational components. Instancename is the name of the new cell instance.

```
procedure compose(pointer(cellDesc) acell;
integer cside, aside;
optional pointer(transform) ptrans;
optional string instancename);
```

executeComposition
Perform the specified composition

```
procedure executeComposition;
```

Predefined transforms are:

unityTransform Null Transform

mirrorXTransform
Mirror the cell over the Y axis (all x coordinates become negative).

mirrorYTransform
Mirror the cell over the X axis (all y coordinates become negative).

rot180Transform Rotate the cell 180 degrees.

rot90Transform Rotate the cell 90 degrees.

rot270transform Rotate the cell 270 degrees.

rot90MXTransform
Rotate the cell 90 degrees and then mirror it over the y axis.

rot270MXTransform
Rotate the cell 270 degrees and then mirror it over the y axis.

As a composition is being executed, there are defaults for both the ports and for the mapping of the sides. Pooh's default is to retain those ports that were not matched with other ports, and to flush those that were. Mapping of the predefined sides also has a default in pooh based on the transformations used while composing the cells. All defaults may be overridden with procedure calls occurring between the original compose specification and the executeComposition command.

The modification commands are:

flushPort Flush the specified port from the composition context.

```
procedure flushPort(boolean add; string
internalName; integer side, position);
```

expungePort Get rid of a port from the current composition context immediately - do not wait until executeComposition. NOTE: expungePort may also be called after executeComposition (and before another compose) to remove ports from the current composition context 'after the fact'.

```
procedure expungePort(string internalName;
integer side, position);
```

omitPort — Omit the specified port from this composition.

```
procedure omitPort(boolean add; string
internalName; integer side, position;
optional string newinternalName; optional
integer newSide, newPosition);
```

keepPort — Keep port in the current composition context, even though it may have been matched up once.

```
procedure keepPort(boolean add; string
internalName; integer side, position;
optional string newinternalName; optional
integer newSide, newPosition);
```

connectPort — Include in this composition a port on a side different then the current composition side.

```
procedure connectPort(boolean add; string
internalName; integer side, position);
```

mapSide — Map from the old side addSide to newSide, after the composition is complete.

```
procedure mapSide(integer addSide, newSide);
```

assignSignal — Assign a signal to the specified port. This will not add the signal to the original cell definition port, but rather to the cell in the current composition context.

```
procedure assignSignal(boolean add;
string internalName;
integer side,position;pointer(signal)
signalname);
```

3.4. Moving Ports

Before performing a composition, it is sometimes desirable to move an old port to a new position. In Pooh, there are two ways to do this: 1) translate an old port to a new position, and 2) add a port to a user defined path (similar to the command 'addpt' for a node). Ports are addressed in the same manner as described at the beginning of this chapter, with one exception; the addportxy addresses a port by an explicit x,y value.

Translate Ports

All the translate port commands have the following format:

```
procedure <procname> (string intName; integer side, pos;
<placement parameters>; integer ptype; optional real radius);
```

<Procname> is the procedure name; IntName, side, pos are the parameters for addressing the port; <Placement parameters> are the placement information; Ptype is the type of path to be used, and radius is the radius of the path. Only the <procname> and <placement parameters> are shown in the following command descriptions.

translatePortAxy(real x, y);
Move to an absolute x,y coordinate.

translatePortAx(real x);
Move to an absolute x, using the original port's y.

translatePortAy(real y);
Move to an absolute y, using the original port's x.

translatePortDxy(real dx, dy);
Move relative to the original port's x and y value.

Add a Port to a Path

These procedures may be used along with multiple calls to initialCell (see initialCell above) to implement a 'place and wire up' form of composition that does not involve matching ports by abutment. This mode of operation requires that absolute positions of the ports be known as well as absolute positions of the placed cells.

addPort — add the addressed port to a user defined path, using the normal port addressing mode.

```
pointer(node) procedure addPort(string intName;
integer side, pos;
optional real dis);
```

addPortXY — add the port whose x and y values are x,y to a user defined path.

```
pointer(node) procedure addPortXY(real x, y;
optional real dis);
```

4. Pooh Status Commands

There are numerous commands available to both query pooh for information about various constructs and to inform pooh about the user's intent. Most of these commands were developed to support the Silicon Compiler project in SSP, but they are available to any pooh user.

These commands fall into five categories: 1) intermediate file directives, 2) design rule information, 3) response file directives, 4) mossim information, and 5) pooh information.

4.1. Intermediate Files

Pooh can write a cell description to two different intermediate files when an enddefine command occurs. The user has control over which of the file types pooh uses. The first file type is CIF (Caltech Intermediate Form) which contains the geometry of a cell. The second file type is ELF (Experimental Layout Format) which contains the connectivity of a cell. Once a file name has been specified, pooh continues to use the same file until either the file is closed, or another file is opened. There are two predefined bits constants which define which file type to use. In all cases, the default file type is CIF. A CIF file has a default file extension of '.CIF', an ELF file has a default file extension of '.ELF'.

Constants

usecif - use a cif file

useelf - use an elf file

openoutputfile use name as the name of the new cif/elf file and close the old output file, if it exists.

```
procedure openoutputfile(string name;
optional bits whichfile);
```

closeoutputfile Close the output file.

```
procedure closeoutputfile(
optional bits whichfile);
```

4.2. Design Rules

The pooh design rules are based on lambda to which the user has access. The user may also ask pooh for the design rules.

getlambda get the current value of lambda

```
real procedure getlambda;
```

setlambda — set the current value of lambda to newlambda.

```
procedure setlambda(real newlambda);
```

nodelayers — find which layers can legally connect to the pooh node p (contained in a port record). Predefined bit constants in nMOS are diffcon for diffusion, polycon for poly, and metalcon for metal.

```
bits procedure nodelayers(pointer(ptnode)p);
```

minwidth — Get the minimum width of the path type layer. If layer is a transistor layer, then get the minimum width of actualtranlayer, which defaults to the centerline layer.

```
real procedure minwidth(integer layer;
optional integer actualtranlayer);
```

minspacing — Get the minimum spacing between two path types. If one or both are transistor path types, then the specific layer(s) may be given in tranlayer1 and tranlayer2. In both cases, the default is the centerline layer.

```
real procedure minspacing(integer layer1,layer2;
optional integer tranlayer1,tranlayer2);
```

4.3. Response File

Pooh maintains a response file to which to write both error messages and information. Initially this file is set to 'tty', but the user may redirect it. A user may also write messages to this response file. If the response file is redirected, the user may choose to have messages go to both 'tty' and to the file.

setresponsefilename — Set the name of the response file. If another response file is open, it will be closed. If ttyalso is true, then a message will also appear on the tty.

```
procedure setresponsefilename(string name;
optional boolean ttyalso);
```

closeresponsefile — Close the current response file and set it to tty.

```
procedure closeresponsefile;
```

writeerror — write msg to the response file, prefixing it with 'ERROR: ', and ending it with an eol.

```
procedure writeerror(string msg);
```

writewarning — write msg to the response file, prefixing it with 'WARNING: ', and ending it with an eol.

```
procedure writewarning(string msg);
```

writeinfo — write information to the response file.

```
procedure writeinfo(repeatable string msg);
```

4.4. Mossim Information

Pooh interfaces to Mossim II, a switch level simulator written at Caltech by Randy Bryant and Mike Schuster. Mossim has certain models of what transistors and electrical nodes look like (refer to the Mossim Manual). The user may set nodes to predefined Mossim values.

pathsize — Set the size of the non-transistor path p to the given size. Predefined sizes are 'small' and 'big'.

```
procedure pathsize(pointer(path)p;
integer size);
```

pathtranstrength — Set the strength of the transistor path p to the given strength. Predefined strengths are 'weak' and 'strong'.

```
procedure pathtranstrength(pointer(path)p;
integer strength);
```

nodetranstrength — Set the strength of the transistor node n to the given strength. Predefined strengths are 'weak' and 'strong'.

```
procedure nodetranstrength(pointer(node)n;
integer strength);
```

4.5. Pooh Cell Information

The Pooh system provides a set of commands which allow a user to query pooh for information. This includes a command which tells pooh to prompt for a response each time a pooh error occurs.

gointodebug — Each time an error message occurs, pooh will ask for a response. This is strictly for debugging.

```
procedure gointodebug;
```

portposition — Given the cell c, and the normal addressing parameters for a port, return the actual position of the port in the cell.

```
pointer(point) procedure portposition(
pointer(celldesc)c;' string internalname;
integer side,position);
```

abutbox — Return the rectangle defined by the cell c's ports. This is the abutment box of c, and indicates the box that other cells may abut to.

```
pointer(rectangle) procedure abutbox(
pointer(celldesc)c);
```

nextp — Return each successive port of the cell c. A call to nextp with firstpoint equal to 'true' will return the first port of c. Each successive call, with firstpoint omitted, will return the next port. Nextp will return a nullpointer when the port list is exhausted.

```
pointer(port) procedure nextp(
optional pointer(celldesc)c;
optional boolean firstpoint);
```

5. Using Pooh On The 20

To use pooh, you must create a mainsail module which includes the header file for the desired technology. This is either "poohnmos.h" or "poohcmos.h". The first command executed by pooh must be a command to set up the design rules. This command is either "setnmos", or "setcmos".

The Pooh Modules are in the library <ssp.lib>pooh.lib. These can either be preloaded before you execute your module, or the command file "pooh.mic" will do it for you. To invoke it, simply type:

```
@do pooh <yourModuleName>
```

Index

Abutbox 33
Addinode 13
Addinxy 14
Additran 14
Addnode 14
AddPort 29
AddPortXY 29
Addpt 11
AssignSignal 28
Ax 10
Axy 11
Ay 10

Baddinode 14
Baddpt 12
Bax 12
Baxy 12
Bay 12
Bdx 12
Bdxy 12
Bdy 12
Binode 14

Closeoutputfile 20
Closeresponsefile 21
CMOS contact Types 6
CMOS path Types 6
Compose 26
ConnectPort 28

DefineCell 8
Dx 10
Dxy 10
Dy 10

EndDefine 8
EndSymbol 9
ExecuteComposition 16
ExpungePort 27

FlushPort 27

Getlambda 30
GetSignal 24
Gointodebug 33

InitialCell 26
Inode 13

KeepPort 28

MakePort 26
MapSide 28
Minspacing 31

Minwidth 31

Name 14
NewP 8
NewSignal 24
NewT 8
Nextp 33
NMOS contact Types 5
NMOS path Types 5
Nodelayers 31
Nodetranstrength 32

OmitPort 27
Openoutputfile 20
Output File Constants 30

Pathsize 22
Pathtranstrength 32
PlaceSymbol 9
Port Addressing 16
Port Types 24
Portposition 23
Predefined Relative Positions 25
Predefined Sides 25
Predefined Signals 23
Predefined Transforms 27
Pullup 15
PullupXY 15

Saddpt 11
Sax 11
Saxy 11
Say 11
Sbaddpt 13
Sbax 13
Sbaxy 13
Sbay 13
Sbdx 13
Sbdxy 13
Sbdy 13
Sdx 11
Sdxy 11
Sdy 11
Setcmos 33
Setlambda 30
Setnmos 33
Setresponsefilename 21

Tran 13
TranslatePortAx 29
TranslatePortAxy 29
TranslatePortAy 29
TranslatePortDxy 29

Writeerror 21
Writeinfo 32
Writewarning 32